Inleiding tot Design Patterns

Head First Design Patterns

Eric Freeman & Elisabeth Freeman with Kathy Sierra and Bert Bates

O' Reilly

Inleiding

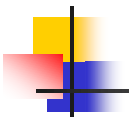
Design Patterns:

Iemand heeft onze problemen al opgelost. We leren waarom en hoe we de kennis en ervaring kunnen inzetten van andere ontwikkelaars die dezelfde weg bij het ontwerpprobleem hebben afgelegd en deze trip hebben overleefd.

Inleiding

Design Patterns:

- **De beste manier om patterns te gebruiken is ze uit het hoofd te leren, om vervolgens de plaatsen waar we ze kunnen toepassen te herkennen in onze ontwerpen en in bestaande applicaties.**
- **We krijgen hergebruik van ervaring.**

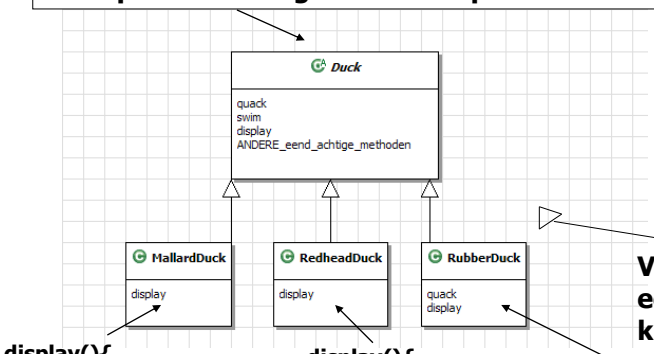


Eerste voorbeeld

We dienen een simulatiespel van een eendenvijver te maken. Het spel kan een groot aantal verschillende eendensoorten tonen die rondzwemmen en kwaken.

Alle eenden kwaken en zwemmen,
De superklasse zorgt voor de implementatiecode

OOAD



Veel andere soorten eenden ervan van de klasse Duck

display(){
//Lijkt op een wilde eend
}

display(){
//Lijkt op een roodkuifeend
}

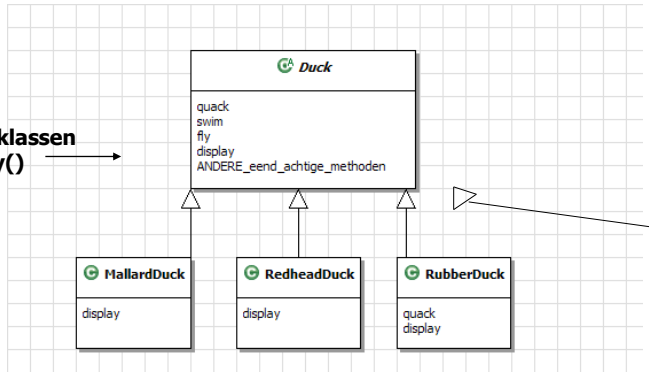
Badeenden kwaken niet, dus wordt quack() overriden door Squeek
display(){
//Lijkt op een badeend
}

Iedere subklasse van Duck is verantwoordelijk voor de implementatie van het gedrag van zijn eigen display() om te laten zien hoe deze op het scherm verschijnt.

Nu moeten de eenden leren vliegen

OOAD

Alle subklassen erven fly()



Andere eendensoorten

Opgelost?

Maar er liep iets helemaal mis. Er vlogen badeenden over het scherm.

Door fly() in de superklasse te plaatsen, kunnen ALLE eenden vliegen, inclusief diegenen die dat niet mogen.

We denken over overerving na …

Oplossing:

RubberDuck

quack() { // squeek }
 	
display() { // badeend }
 	
fly() {
  // override, zodat ze niks doen
}

Maar wat gebeurt er dan wanneer we houten lokeenden aan het programma toevoegen? Die vliegen en kwaken ook niet.

DecoyDuck

quack() {// override, zodat ze niks doen}
 	
display() { // houten lokeend }
 	
fly() {
  // override, zodat ze niks doen
}

8

Welke van de volgende alternatieven zijn nadelig voor het gebruik van overerving om voor het gedrag van een eend te zorgen?

A. Code wordt in de subklassen gedupliceerd.

B. Runtime gedragsveranderingen zijn zéér moeilijk/onmogelijk

C. We kunnen eenden niet laten dansen.

D. Het is moeilijk om het gedrag van alle eenden te kennen.

E. Eenden kunnen niet tegelijk vliegen en kwaken

F. Veranderingen kunnen onbedoeld andere eenden beïnvoeden.

9

De specificaties blijven veranderen. Iedere keer er een nieuwe subklasse aan Duck wordt gehangen, zijn we gedwongen naar de methoden fly() en quack() te kijken of deze via een override vervangen moeten worden.

→ Hergebruik door overerving, hetgeen met het oog op onderhoud NIET zo goed uitpakte.

10

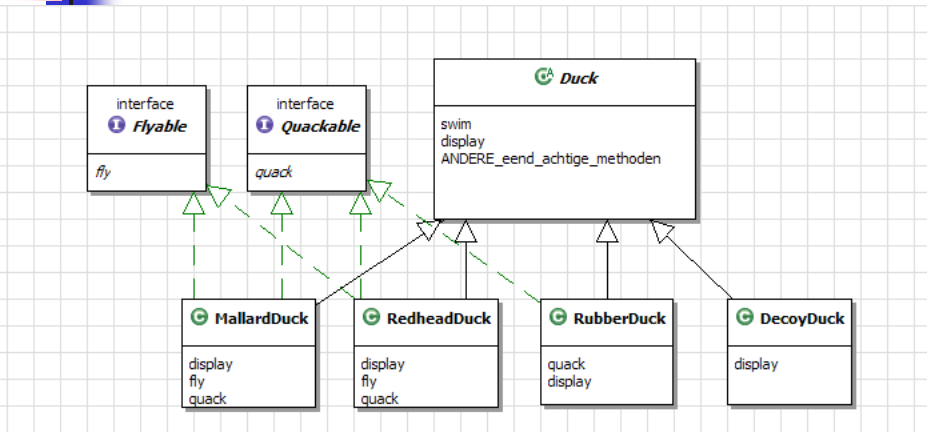
De enige constante bij softwareontwikkeling

Los van wat je maakt of in welke taal je programmeert, de enige constante die je altijd tegenkomt is

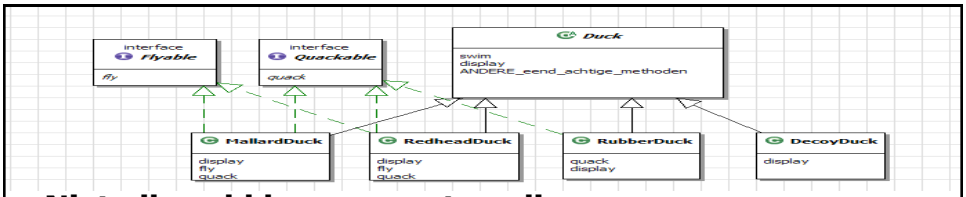
VERANDERING

Interface

We hebben een betere manier nodig waarbij slechts een aantal (maar niet alle) eendensoorten vliegen en kwaken.



Wat denk je van dit ontwerp?



- **Niet alle subklassen moeten vlieg- en kwaakgedrag vertonen → overerving is niet het juiste antwoord.**

- **Via de implementatie van `Flyable` en `Quackable` in de subklassen is een deel van het probleem opgelost → het concept van hergebruik van code voor een dergelijk gedrag wordt onmogelijk! We creëren een andere nachtmerrie voor het onderhoud! Bovendien kunnen er meerdere soorten vlieggedrag voorkomen bij eenden die wel kunnen vliegen …**

- **We gaan een oplossing maken … door de juiste softwareontwerpprincipes volgens OO toe te passen.**

OOAD

Ontwerpprincipe

Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven.

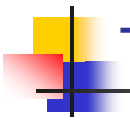
M.a.w. bevat onze code een aspect dat verandert, bv. met ieder nieuwe systeemeis, dan weten we dat we een gedrag hebben dat eruit gelicht moet worden en moet worden afgezonderd van alle code die niet verandert.

14

Het eendengedrag ontwerpen

- **Flexibel blijven**
- **Gedrag aan de instanties van Duck toekennen**

- **We willen (nu we toch bezig zijn) het gedrag van een eend dynamisch veranderen. M.a.w. we moeten methoden voor het instellen van het gedrag in de klassen van Duck opnemen, zodat we het gedrag van een subklasse dynamisch kunnen veranderen. Bv. het vlieggedrag van de klasse MallardDuck at runtime kunnen veranderen.**



Tweede ontwerpprincipe

Programmeer naar een interface, niet naar een implementatie.

M.a.w. we gaan voor ieder gedrag een interface gebruiken, bv. FlyBehavior en QuackBehavior, en iedere implementatie van een gedrag implementeert één van deze interfaces.

Van nu af aan staat het gedrag van Duck in een afzonderlijke klasse – een klasse die een bepaalde gedragsinterface implementeert.

Op deze manier hoeven de klassen van Duck niks te weten van de implementatiedetails van hun eigen gedrag.

Interface

- Het woord "interface" heeft een dubbele betekenis.

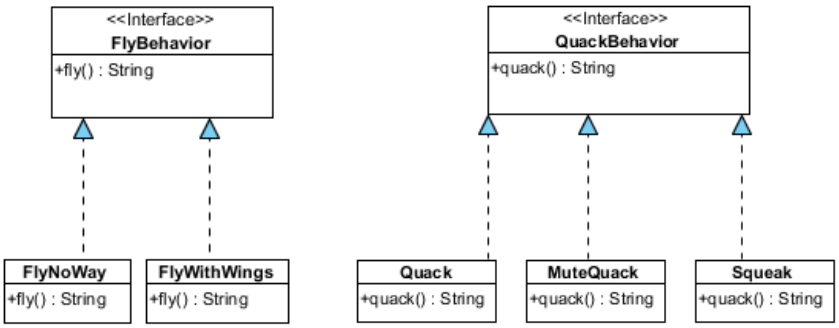
concept "interface" — constructie met de naam "interface" in JAVA.

- **Je kunt naar een interface programmeren zonder de interface van JAVA te gebruiken. Het gaat er om zodanig van polymorfisme gebruik te maken door naar een supertype te programmeren zodat het actuele runtimeobject niet is opgesloten in de code.**

een abstracte klasse — **interface** — **een concrete klasse**

17

Implementatie van het eendengedrag



- **In dit ontwerp kunnen andere objecttypen ons gedrag voor vliegen en kwaken hergebruiken omdat dit gedrag niet langer binnen onze klassen van Duck is weggeborgen.**

- **En we kunnen nieuw gedrag toevoegen zonder dat we de bestaande gedragsklassen hoeven te veranderen of de klassen van Duck die het vlieggedrag gebruiken hoeven te benaderen.**

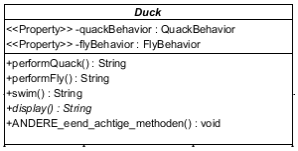
Integratie van het eendengedrag

OOAD

- **Duck delegeert het vlieg- en kwaakgedrag!**
- **Dat gaat als volgt:**

1. **Eerst voegen we twee instantievariabelen toe**

```
FlyBehavior flyBehavior
QuackBehavior quackBehavior
```



2. **We implementeren performQuack()**

```
public abstract class Duck
{
        private QuackBehavior quackBehavior;
        …
        public String performQuack()
        {
                return quackBehavior.quack();
        } …
```

Om te kwaken staat Duck het object dat wordt aangewezen door quackBehavior, toe om voor hem te kwaken.
In dit deel van de code kan het ons niet schelen welk soort object het is, ons interesseert alleen maar dat het weet hoe het quack() uitvoert.

Nog meer integratie …

OOAD

- **Hoe worden de instantievariabelen flyBehavior en quackBehavior ingesteld?**
- **Bv. klasse MallardDuck:**

```
public class MallardDuck extends Duck {

        public MallardDuck()
        {
                setQuackBehavior(new Quack());
                setFlyBehavior(new FlyWithWings());
        }

        public String display()
        {
                return "Ik ben een echte wilde eend";
        }

}
```

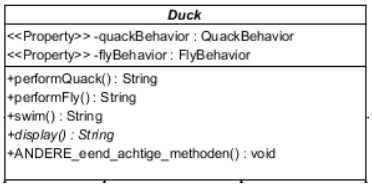
Wacht eens even,

```
public MallardDuck()
{
        setQuackBehavior(new Quack());
        setFlyBehavior(new FlyWithWings());
}
```

- **We mogen NIET naar een implementatie programmeren. Maar in de constructor maken we een nieuwe instantie van een concrete implementatie van de klasse Quack!**
  → dat doen we voor het moment. We zullen nog meer patterns leren waarmee we dit kunnen verhelpen.

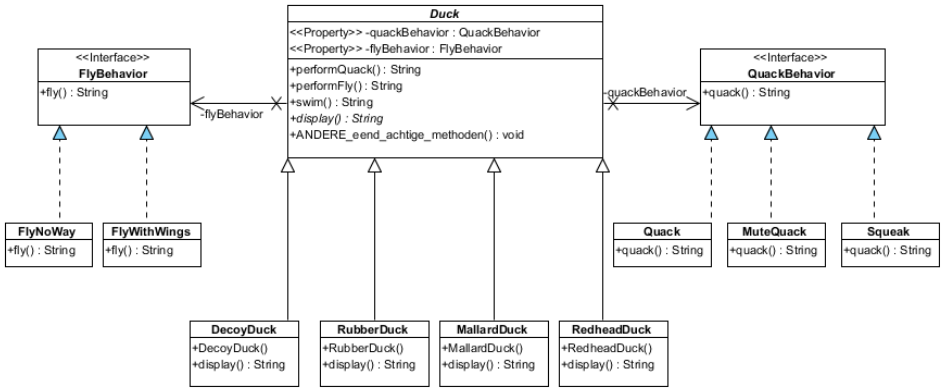
Positief:

- **Veel flexibiliteit**
- **We kunnen het gedrag van een eend at runtime veranderen**




```
public void setFlyBehavior(FlyBehavior flyBehavior)
{
        this.flyBehavior = flyBehavior;
}

public void setQuackBehavior(QuackBehavior quackBehavior)
{
        this.quackBehavior = quackBehavior;
}
```



- **Ingekapseld vlieggedrag:**
  familie algoritmen (verzameling gedragingen)
- **Ingekapseld kwaakgedrag**
- **Client: de client gebruikt een ingekapselde**

familie algoritmen voor zowel het vliegen als het kwaken

23

HEEFT-EEN is soms beter dan IS-EEN

OOAD

De relatie HEEFT-EEN:
iedere eend heeft een FlyBehavior en een QuackBehavior waaraan de eend het vliegen en kwaken delegeert.

We gebruiken COMPOSITIE (composition + aggregation).
In plaats van het gedrag te erven, verkrijgt de eend zijn gedrag uit het (juiste) gedragsobject waarmee hij is samengesteld.

Ontwerpprincipe

Geef aan compositie de voorkeur boven overerving.

Het creëren van systemen via compositie geeft meer flexibiliteit. We kunnen een familie algoritmen inkapselen. We kunnen tevens het gedrag at runtime veranderen.
Compositie wordt in veel design patterns toegepast.

Eerste design pattern

Strategy – gedrag van objecten

Het Strategy Pattern definieert een familie algoritmen, isoleert ze en maakt ze uitwisselbaar. Strategy maakt het mogelijk om het algoritme los van de client die deze gebruikt, te veranderen